1

Introduction to XML

COPYRIGHTED MATERIAL

Extensible Markup Language (XML) is a language defined by the World Wide Web Consortium (W3C, `http://www.w3c.org`), the body that sets the standards for the Web. You can use XML to create your own elements, thus creating a customized markup language for your own use. In this way, XML supersedes other markup languages such as Hypertext Markup Language (HTML); in HTML, all the elements you use are predefined — and there are not enough of them. In fact, XML is a metamarkup language because it lets you create your own markup languages.

XML is the next logical step in developing the full potential of the Internet and the Web. Just as HTML, HyperText Transfer Protocol (HTTP), and Web browsers paved the way for exciting new methods of communications between networked computers and people, XML and its associated technologies open new avenues of electronic communications between people and machines. In the case of XML, however, the promise is for both human-machine and machine-machine communications, with XML as the "lowest-common-denominator" language that all other systems — proprietary or open — can use.

XML derives much of its strength in combination with the Web. The Web provides a collection of protocols for moving data; XML represents a way to define that data. The most immediate effect has been a new way to look at the enterprise. Instead of a tightly knit network of servers, the enterprise is now seen as encompassing not just our traditional networks but also the Web itself, with its global reach and scope. XML has become the unquestionable standard for generically marking data to be shared. As XML continues to grow in popularity, so too are the number of ways in which XML is being implemented. XML can be used for a variety of purposes, from obvious tasks such as marking up simple data files and storing temporary data to more complex tasks such as passing information from one program or process to another.

XML finds several applications in business and, increasingly, in everyday life. It provides a common data format for companies that want to exchange documents. It's used by Web services to encode messages and data in a platform-independent manner. It's even used to build Web sites, where it serves as a tool for cleanly separating content from appearance.

This chapter is about XML as a language and its related technologies. A comprehensive treatment of the subject could easily fill 300 pages or more, so this chapter attempts to strike a reasonable balance between detail and succinctness. In the pages that follow, you learn about the different XML-related technologies and their usage. But before that, take a brief look at XML itself.

A Primer on XML

XML is derived from the Standard Generalized Markup Language (SGML), a rich language used mostly for huge documentation projects. The designers of XML drew heavily from SGML and were guided by the lessons learned from HTML. They produced a specification that was only about 20 percent the size of the SGML specification, but nearly as powerful. Although SGML is typically used by those who need the power of an industrial-strength language, XML is intended for everyone.

One of the great strengths of XML is the extensibility it brings to the table. XML doesn't have any tags of its own and it doesn't constrain you like other markup languages. Instead, XML defines rules for developing semantic tags of your own. The tags you create form vocabularies that can be used to structure data into hierarchical trees of information. You can think of XML as a metamarkup language that enables developers, companies, and even industries to create their own, specific markup languages.

One of the most important concepts to grasp in XML is about content, not presentation. The tags you create focus on organizing your data rather than displaying it. XML isn't used, for example, to indicate a particular part of a document in a new paragraph or that another part should be bolded. XML is used to develop tags that indicate a particular piece of data is the author's first name, another piece is the book title, and a third piece is the published year of the book.

Self-Describing Data

As mentioned before, the most powerful feature of XML is that it doesn't define any tags. Creating your own tags is what makes XML extensible; however, defining meaningful tags is up to you. When creating tags, it isn't necessary to abbreviate or shorten your tag names. It doesn't make processing them any faster. but it can make your XML documents more confusing or easier to understand. Remember, developers are going to be writing code against your XML documents. On the one hand, you could certainly define tags like the following:

```
<H1>XSLT Programmers Reference
<p><b>Michael Kay</b></p>
</H1>
```

Using these HTML-based tags might make it easy to be displayed in a browser, but they don't add any information to the document. Remember, XML is focused on content, not presentation. Creating the following XML would be far more meaningful:

```
<books>
  <book>
    <title>XSLT Programmers Reference</title>
    <author>Michael Kay</author>
  </book>
</books>
```

The second example is far more readable in human terms, and it also provides more functionality and versatility to nonhumans. With this set of tags, applications can easily access the book's title or author name without splitting any strings or searching for spaces. And, for developers writing code, searching for the author name in an XML document becomes much more natural when the name of the element is title, for example, rather than H1.

> *Indenting the tags in the previous example was done purely for readability and certainly isn't necessary in your XML documents. You may find, however, when you create your own documents, indentation helps you to read them.*

To process the previous XML data, no special editors are needed to create XML documents, although a number of them are available. And no breakthrough technology is involved. Much of the attention swirling around XML comes from its simplicity. Specifically, interest in XML has grown because of the way XML simplifies the tasks of the developers who employ it in their designs. Many of the tough tasks software developers have to do again and again over the years are now much easier to accomplish. XML also makes it easier for components to communicate with each other because it provides a standardized, structured language recognized by the most popular platforms today. In fact, in the .NET platform, Microsoft has demonstrated how important XML is by using it as the underpinning of the entire platform. As you see in later chapters, .NET relies heavily on XML and SOAP (Simple Object Access Protocol) in its framework and base services to make development easier and more efficient.

Basic Terminology

XML terminology is thrown around, sometimes recklessly, within the XML community. Understanding this terminology will help you understand conversations about XML a little more.

Well-Formed

A document is considered to be well-formed if it meets all the well-formedness constraints defined by XML specification. These constraints are as follows:

- The document contains one or more elements.
- The document consists of exactly one root element (also known as the document element).
- The name of an element's end tag matches the name defined in the start tag.
- No attribute may appear more than once within an element.
- Attribute values cannot contain a left-angle bracket (<).
- Elements delimited with start and end tags must nest properly within each other.

Validity

First and foremost, a valid XML document must be well-formed before it can even think about being a valid XML document. The well-formed requirement should be fairly straightforward, but the key that makes an XML document leap from well-formed to valid is slightly more difficult. To be valid, an XML document must be validated. A document can be validated through a Document Type Definition (DTD), or an XML Schema Definition (XSD). For the XML document to be valid, it must conform to the constraints expressed by the associated DTD or the XSD schema.

A valid document does not ensure semantic perfection. Although XML Schema defines stricter constraints on element and attribute content than XML DTDs do, it cannot catch all errors. For example, you might define a price datatype that requires two decimal places; however, you might enter 1600.00 when you meant to enter 16.00, and the schema document wouldn't catch the error.

When dealing with validity, you need to keep in mind that there are three ways an XML document can exist:

- As a free-form, well-formed XML document that does not have DTD or schema associated with it
- As a well-formed and valid XML document, adhering to a DTD or schema
- As a well-formed document that is not valid because it does not conform to the constraints defined by the associated DTD or schema

Now that you have a general understanding of the XML concepts, the next section examines the constituents of an XML document.

Components of an XML Document

As mentioned earlier in this chapter, XML is a language for describing data and the structure of data. XML data is contained in a document, which can be a file, a stream, or any other storage medium, real or virtual, that's capable of holding text. A proper XML document begins with the following XML declaration, which identifies the document as an XML document and specifies the version of XML that the document's contents conform to:

```
<?xml version="1.0"?>
```

The XML declaration can also include an encoding attribute that identifies the type of characters contained in the document. For example, the following declaration specifies that the document contains characters from the Latin-1 character set used by Windows 95, 98, and Windows Me:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

The next example identifies the character set as UTF-16, which consists of 16-bit Unicode characters:

```
<?xml version="1.0" encoding="UTF-16"?>
```

The encoding attribute is optional if the document consists of UTF-8 or UTF-16 characters because an XML parser can infer the encoding from the document's first five characters: `'<?xml'`. Documents that use other encodings must identify the encodings that they use to ensure that an XML parser can read them. XML declarations are actually specialized forms of XML processing instructions that contain commands for XML processors. Processing instructions are always enclosed in `<?` and `?>` symbols. Some browsers, such as Internet Explorer, interpret the following processing instruction to mean that the XML document should be formatted using a style sheet named `Books.xsl` before it's displayed:

```
<?xml-stylesheet type="text/xsl" href="Books.xsl"?>
```

The XML declaration is followed by the document's root element, which is usually referred to as the document element. In the following example, the document element is named books:

```
<?xml version="1.0"?>
<books>
  ...
</books>
```

The document element is not optional; every document must have one. The following XML is legal because book elements are nested within the document element books:

```
<?xml version="1.0"?>
<books>
  <book>
    ...
  </book>
  <book>
    ...
  </book>
</books>
```

The document in the next example, however, is not legal because it lacks a document element:

```
<?xml version="1.0"?>
<book>
  ...
</book>
<book>
  ...
</book>
```

If you run the previous XML through a parser, the XML will not load properly, complaining about the non-existence of the root element.

Elements

Element names conform to a set of rules prescribed in the XML specification that you can read at `http://www.w3.org/TR/REC-xml`. The specification essentially says that element names can consist of letters or underscores followed by letters, digits, periods, hyphens, and underscores. Spaces are not permitted in element names. Elements are the building blocks of XML documents and can contain data, other elements, or both, and are always delimited by start and end tags. XML has no predefined elements; you define elements as needed to adequately describe the data contained in an XML document. The following document describes a collection of books:

```
<?xml version="1.0"?>
<books>
  <book>
    <title>XSLT Programmers Reference</title>
    <author>Michael Kay</author>
    <year>2003</year>
  </book>
  <book>
```

```
    <title>ASP.NET 2.0 Beta Preview</title>
    <author>Bill Evjen</author>
    <year></year>
  </book>
</books>
```

In this example, `books` is the document element, book elements are children of books, and title, and author are children of book. The book elements contain no data (just other elements), but title, and author contain data. The following line in the second book element contains neither data nor other elements.

```
<year></year>
```

Empty elements are perfectly legal in XML. An empty year element can optionally be written this way for conciseness:

```
<year/>
```

Unlike HTML, XML requires that start tags be accompanied by end tags; therefore, the following XML is never legal:

```
<year>2003
```

Also unlike HTML, XML is case-sensitive. A `<year>` tag closed by a `</ Year>` tag is not legal because the cases of the `Y`s do not match.

Because XML permits elements to be nested within elements, the content of an XML document can be viewed as a tree. By visualizing the document structure in a tree, you can clearly understand the parent-child relationships among the document's elements.

Attributes

XML allows you to attach additional information to elements by including attributes in the elements' start tags. Attributes are name/value pairs. The following book element expresses year as an attribute rather than as a child element:

```
<book year="2003">
  <title>XSLT Programmers Reference</title>
  <author>Michael Kay</author>
</book>
```

Attribute values must be enclosed in single or double quotation marks and may include spaces and embedded quotation marks. (An attribute value delimited by single quotation marks can contain double quotation marks and vice versa.) Attribute names are subject to the same restrictions as element names and therefore can't include spaces. The number of attributes an element can be decorated with is not limited.

> **When defining a document's structure, it's sometimes unclear — especially to XML newcomers — whether a given item should be defined as an attribute or an element. In general, attributes should be used to define out-of-band data and elements to define data that is integral to the document. In the previous example, it probably makes sense to define year as an element rather than an attribute because year provides important information about the book in question.**

Now consider the following XML document:

```
<book image="xslt.gif">
  <title>XSLT Programmers Reference</title>
  <author>Michael Kay</author>
  <year>2003</year>
</book>
```

The image attribute contains additional information that an application might use to display the book information with a picture. Because no one other than the software processing this document is likely to care about the image, and because the image is an adjunct to (rather than a part of) the book's definition, image is properly cast as an attribute instead of an element.

CDATA, PCDATA, and Entity References

Textual data contained in an XML element can be expressed as Character Data (CDATA), Parsed Character Data (PCDATA), or a combination of the two. Data that appears between `<![CDATA[` and `]]>` tags is CDATA; any other data is PCDATA. The following element contains PCDATA:

```
<title>XSLT Programmers Reference</title>
```

The next element contains CDATA:

```
<author><![CDATA[Michael Kay]]></author>
```

And the following contains both:

```
<title>XSLT Programmers Reference <![CDATA[Author - Michael Kay]]></title>
```

As you can see, CDATA is useful when you want some parts of your XML document to be ignored by the parser and not processed at all. This means you can put anything between `<![CDATA[` and `]]>` tags and an XML parser won't care; however data not enclosed in `<![CDATA[` and `]]>` tags must conform to the rules of XML. Often, CDATA sections are used to enclose code for scripting languages like VBScript or JavaScript.

XML parsers ignore CDATA but parse PCDATA — that is, interpret it as markup language. You might wonder why an XML parser distinguishes between CDATA and PCDATA. Certain characters, notably `<`, `>`, and `&`, have special meaning in XML and must be enclosed in CDATA sections if they're to be used verbatim. For example, suppose you wanted to define an element named range whose value is `'0 <` counter `< 1000'`. Because `<` is a reserved character, you can't define the element this way:

```
<range>0 < counter < 1000</range>
```

You can, however, define it this way:

```
<range><[CDATA[0 < counter < 100]]></range>
```

As you can see, CDATA sections are useful for including mathematical equations, code listings, and even other XML documents in XML documents.

Another way to include <, >, and & characters in an XML document is to replace them with entity references. An entity reference is a string enclosed in & and ; symbols. XML predefines the following entities:

Symbol	Corresponding Entity
<	lt
>	gt
&	amp
'	apos
"	quot

Using the entity references, you can alternatively define a range element with the value `'0 < counter < 1000'`:

```
<range>0 &lt; counter &lt; 100</range>
```

You can also represent characters in PCDATA with character references, which are nothing more than numeric character codes enclosed in `&#` and `;` symbols, as in

```
<range>0 &#60; counter &#60; 100</range>
```

Character references are useful for representing characters that can't be typed from the keyboard. Entity references are useful for escaping the occasional special character, but for large amounts of text containing arbitrary content, CDATA sections are far more convenient.

Namespaces

A namespace groups elements together by partitioning elements and their attributes into logical areas and providing a way to identify the elements and attributes uniquely. Namespaces are also used to reference a particular DTD or XML Schema. Namespaces were defined after XML 1.0 was formally presented to the public. After the release of XML 1.0, the W3C set out to resolve a few problems, one of which is related to naming conflicts. To understand the significance of this problem, first think about the future of the Web.

Shortly after the W3C introduced XML 1.0, an entire family of languages such as Mathematical Markup Language (MathML), Synchronized Multimedia Integration Language (SMIL), Scalable Vector Graphics (SVG), XLink, XForms, and the Extensible Hypertext Markup Language (XHTML) started appearing. Instead of relying on one language to bear the burden of communicating on the Web, the idea was to present many languages that could work together. If functions were modularized, each language could do what it does best; however the problem arises when a developer needs to use multiple vocabularies within the same application. For example, one might need to use a combination of languages such as SVG, SMIL, XHTML, and XForms for an interactive Web site. When mixing vocabularies, you have to have a way to distinguish between element types. Take the following example:

```
<html>
  <head>
    <title>Book List</title>
  </head>
```

```
    <body>
      <books>
        <book>
          <title>XSLT Programmers Reference</title>
           <author>Michael Kay</author>
        </book>
      </books>
    </body>
  </html>
```

In this example, there's no way to distinguish between the two title elements even though they are semantically different. A namespace can solve this problem by providing a unique identifier for a collection of elements and/or attributes. This is accomplished by prefixing each member element and attribute with a name, uniquely identifying them as part of that namespace. Grouping elements into a namespace allows them to be referenced easily by many XML documents and allows one XML document to reference many namespaces. XML namespaces are a form of qualifying attribute and element names. This is done within XML documents by associating them with namespaces that are identified with Universal Resource Indicators (URIs).

> *A URI is a unique name recognized by the processing application that identifies a particular resource. URIs includes Uniform Resource Locators (URL) and Uniform Resource Numbers (URN).*

The following is an example of using a namespace declaration that associates the namespace `http://www.w3.org/1999/xhtml` with the HTML element.

```
<html xmlns ="http://www.w3.org /1999/xhtml">
```

The `xmlns` keyword is a special kind of attribute that indicates you are about to declare an XML namespace. The information between the quotes is the URI, pointing to the actual namespace — in this case, a schema. The URI is a formal way to differentiate between namespaces; it doesn't necessarily need to point to anything at all. The URI is used only to demarcate elements and attributes uniquely. The `xmlns` declaration is placed inside the element tag using the namespace.

> *Namespaces can confuse XML novices because the namespace names are URIs and therefore often mistaken for a Web address that points to some resource; however, XML namespace names are URLs that don't necessarily have to point to anything. For example, if you visit the XSLT namespace (*`http://www.w3.org/1999/XSL/Transform`*), you would find a single sentence: "This is an XML Namespace defined in the XSL Transformations (XSLT) Version 1.0 specification." The unique identifier is meant to be symbolic; therefore, there's no need for a document to be defined. URLs were selected for namespace names because they contain domain names that can work globally across the Internet and they are unique.*

The following code shows the use of namespaces to resolve the name conflict in the preceding example.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Book List</title>
  </head>
  <body>
    <books xmlns="http://www.wrox.com/books/xml">
      <book>
        <title>XSLT Programmers Reference</title>
        <author>Michael Kay</author>
      </book>
```

```
      </books>
    </body>
  </html>
```

The books element belongs to the namespace `http://www.wrox.com/books/xslt`, whereas all the XHTML elements belong to the XHTML namespace `http://www.w3.org/1999/xhtml`.

Declaring Namespaces

To declare a namespace, you need to be aware of the three possible parts of a namespace declaration:

- `xmlns` — Identifies the value as an XML namespace and is required to declare a namespace and can be attached to any XML element.
- `prefix` — Identifies a namespace prefix. It (including the colon) is only used if you're declaring a namespace prefix. If it's used, any element found in the document that uses the prefix (prefix:element) is then assumed to fall under the scope of the declared namespace.
- `namespaceURI` — It is the unique identifier. The value does not have to point to a Web resource; it's only a symbolic identifier. The value is required and must be defined within single or double quotation marks.

There are two different ways you can define a namespace:

- `Default namespace` — Defines a namespace using the xmlns attribute without a prefix, and all child elements are assumed to belong to the defined namespace. Default namespaces are simply a tool to make XML documents more readable and easier to write. If you have one namespace that will be predominant throughout your document, it's easier to eliminate prefixing each of the elements with that namespace's prefix.
- `Prefixed namespace` — Defines a namespace using the `xmlns` attribute with a prefix. When the prefix is attached to an element, it's assumed to belong to that namespace.

> **Default namespaces save time when creating large documents with a particular namespace; however, they don't eliminate the need to use prefixes for attributes.**

The following example demonstrates the use of default namespaces and prefixed namespaces.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Book List</title>
</head>
<body>
  <blist:books
    xmlns:blist="http://www.wrox.com/books/xml">
    <blist:book>
      <blist:title>XSLT Programmers Reference</blist:title>
      <blist:author>Michael Kay</blist:author>
    </blist:book>
  </blist:books>
</body>
</html>
```

The `xmlns` defined at the root HTML element is the default namespace applied for all the elements that don't have an explicit namespace defined; however the books element defines an explicit namespace using the prefix `blist`. Because that prefix is used while declaring the books elements, all of the elements under books are considered to be using the prefixed namespace.

Role of Namespaces

A namespace is a set of XML elements and their constituent attributes. As you dive deep into XML, such as creating interactive XML Web pages for the Web and establishing guidelines for transporting data and so on, you will find that XML namespaces are incredibly important. Here are some of the uses of namespaces.

Reuse

Namespaces can allow any number of XML documents to reference them. This allows namespaces to be reused as needed, rather than forcing developers to reinvent them for each document they create. For instance, consider the common business scenario wherein you have two applications that exchange a common XML format: the server that generates the XML, relying on a particular namespace, and the client that consumes this XML, which also must rely on the same namespace. Rather than generating two namespaces (one for each application), a single namespace can be referenced by both applications in the XML they generate. This enables namespaces to be reused, which is an important feature. Not only can namespaces be reused by different parts of one application, they can be reused by different parts of any number of applications. Therefore, investing in developing a well thought out namespace can pay dividends for some time.

Multiple Namespaces

Just as multiple XML documents can reference the same namespace, one document can reference more than one namespace. This is a natural by-product of dividing elements into logical, ordered groups. Just as software development often breaks large processes into smaller procedures, namespaces are usually chunked into smaller, more logical groupings. Creating one large namespace with every element you think you might need doesn't make sense. This would be confusing to develop and it certainly would be confusing to anyone who had to use such an XML element structure. Rather, granular, more natural namespaces should be developed to contain elements that belong together.

For instance, you can create the namespaces as building blocks, assembled together to form the vocabularies required by a large program. For example, an application might perform services that help users to buy products from an e-commerce Web site. This application would require elements that define product categories, products, buyers, and so on. Namespaces make it possible to include these vocabularies inside one XML document, pulling from each namespace as needed.

Ambiguity

Namespaces can sometimes overlap and contain identical elements. This can cause problems when an XML document relies on the namespaces in question. An example of such a collision might be a namespace containing elements for book orders and another with elements for book inventories. Both might use elements that refer to a book's title or an author's name. When one document attempts to reference elements from both namespaces, this creates ambiguity for the XML parser. You can resolve this problem by wrapping the elements of book orders and book inventories in separate namespaces. Because elements and attributes that belong to a particular namespace are identified as such, they don't conflict with other elements and attributes sharing the same name. This solves the previously mentioned ambiguity. By prefacing a particular element or attribute name with the namespace prefix, a parser can correctly reconcile any potential name collisions. The process of using a namespace prefix creates qualified names for each of the elements and attributes used within a document.

XML Technologies

As the popularity of XML grows, new technologies that complement XML's capabilities also continue to grow. The following section takes a quick tour of the important XML technologies that are essential to the understanding and development of XML-based ASP.NET Web applications.

DTD

One of the greatest strengths of XML is that it allows you to create your own tag names. But for any given application, it is probably not meaningful for any kind of tags to occur in a completely arbitrary order. If the XML document is to have meaning, and certainly if you're writing a style sheet or application to process it, there must be some constraint on the sequence and nesting of tags. DTDs are one way using which constraints can be expressed.

DTDs, often referred to as doctypes, consist of a series of declarations for elements and associated attributes that may appear in the documents they validate. If this target document contains other elements or attributes, or uses included elements and attributes in the wrong way, validation will fail. In effect, the DTD defines a grammar for the documents it validates.

The following shows an example of what a DTD looks like:

```
<?xml version="1.0" ?>
<!-- DTD is not parsed as XML, but read by parser for validation -->
<!DOCTYPE book [
<!ELEMENT book (title, chapter+)>
<!ATTLIST book author CDATA #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT chapter (#PCDATA)>
<!ATTLIST chapter id #REQUIRED>
]>
```

From the preceding DTD, you can already recognize enough vocabulary to understand this DTD as a definition of a book document that has elements book, title, and chapter and attributes author and id. A DTD can exist inline (inside the XML document), or it can be externally referenced using a URL.

A DTD also includes information about data types, whether values are required, default values, number of allowed occurrences, and nearly every other structural aspect you could imagine. At this stage, just be aware that your XML-based applications may require an interface with these types of information if your partners have translated documents from SGML to XML or are leveraging part of their SGML infrastructure.

As mentioned before, DTDs may either be stored internally as part of the XML document or externally in a separate file, accessible via a URL. A DTD is associated with an XML document by means of a `<!DOCTYPE>` declaration within the document. This declaration specifies a name for the `doctype` (which should be the same as the name of the root element in the XML document) along with either a URL reference to a remote DTD file, or the DTD itself.

It is possible to reference both external and internal DTDs, in which case the internal DTD is processed first, and duplicate definitions in the external file may cause errors. To specify an external DTD, use either the SYSTEM or PUBLIC keyword as follows:

```
<!DOCTYPE docTypeName SYSTEM "http://www.wrox.com/Books.dtd">
```

Using SYSTEM as shown allows the parser to load the DTD from the specified location. If you use PUBLIC, the named DTD should be one that is familiar to the parser being used, which may have a store of commonly used DTDs. In most cases, you will want to use your own DTD and use SYSTEM. This method enables the parsing application to make its own decisions as to what DTD to use, which may result in a performance increase; however, specific implementation of this is down to individual parsers, which might limit the usefulness of this technique.

> **As useful as DTDs are, they also have their shortcomings. The major concern most developers have with DTDs is the lack of strong type-checking. Also, DTDs are created using a strange and seemingly archaic syntax. They have only a limited capability in describing the document structure in terms of how many elements can nest within other elements.**

Because of the inherent disadvantages of DTDs, XML Schemas are the commonly used mechanism to validate XML documents. XML schemas are discussed in detail in a later section of this chapter.

XDR

XML Data Reduced (XDR) schema is Microsoft's own version of the W3C's early 1999 work-in-progress version of XSD. This schema is based on the W3C Recommendation of the XML-Data Note (`http://www.w3.org/TR/1998/NOTE-XML-data`), which defines the XML Data Reduced schema.

The following document contains the same information that you could find in a DTD. The main difference is that it has the structure of a well-formed XML document. This example shows the same constraints as the DTD example, but in an XML schema format:

```
<?xml version="1.0" ? >
<!-- XML-Data is a standalone valid document-->
<Schema xmlns="urn:schemas-microsoft-com:xml-data">
  <AttributeType name="author" required="yes"/>
  <AttributeType name="id" required="yes"/>
  <ElementType name="title" content="textOnly"/>
  <ElementType name="chapter" content="textOnly"/>
  <ElementType name="book" content="eltOnly">
    <attribute type="author" />
    <element type="title" />
    <element type="chapter" />
  </ElementType>
</Schema>
```

There are a few things that an XDR schema can do that a DTD cannot. You can directly add data types, range checking, and external references called namespaces.

XSD

The term *schema* is commonly used in the database community and refers to the organization or structure for a database. When this term is used in the XML community, it refers to the structure (or model) of a class of documents. This model describes the hierarchy of elements and allowable content in a valid XML document. In other words, the schema defines constraints for an XML vocabulary.

New standards for defining XML documents have become desirable because of the limitations imposed by DTDs. XML Schema Definition (XSD) schema, sometimes referred to as an XML schema, is a formal definition for defining a schema for a class of XML documents. The sheer volume of text involved in defining the XML schema language can be overwhelming to an XML novice, or even to someone making the move from DTDs to XML schema. As previously stated before our detour into namespaces, XML schemas have evolved as a response to problems with the W3C's first attempt at data validation, DTDs. DTDs are a legacy inherited from SGML to provide content validation and, although DTDs do a good job of validating XML, certainly room does exist for improvement. Some of the more important concerns expressed about DTDs are the following:

- DTD uses Extended Backus Naur Form syntax, which is dissimilar to XML.
- DTDs aren't intuitive, and they can be difficult to interpret from a human-readable point of view.
- The metadata of DTDs is programmatically difficult to consume.
- No support exists for data types.
- DTDs cannot be inherited.

To address these concerns, the W3C developed a new validating mechanism to replace DTDs called XML schemas. Schemas provide the same features DTDs provide, but they were designed with the previous issues in mind and thus are more powerful and flexible. The design principles outlined by the XML Schema Requirements document are fairly straightforward. XML schema documents should be created so they are as follows:

- More expressive than XML DTDs
- Expressed in XML
- Self-describing
- Usable in a wide variety of applications that employ XML
- Straightforwardly usable on the Internet
- Optimized for interoperability
- Simple enough to implement with modest design and runtime resources
- Coordinated with relevant W3C specs, such as XML Information Set, XML Linking Language (XLink), Namespaces in XML, Document Object Model (DOM), HTML, and the Resource Description Framework (RDF) schema

As mentioned earlier in this chapter, an XML schema is a method used to describe XML attributes and elements. This method for describing the XML file is actually written using XML, which provides many benefits over other validation techniques, such as DTD. These benefits include the following:

- Because the schema is written in XML, you don't have to know an archaic language to describe your document. Because you already know XML, using XSD schema is fairly easy and straightforward.
- The same engines to parse XML documents can also be used to parse schemas.
- Just as you can parse schemas in the same fashion as XML, you can also add nodes, attributes, and elements to schemas in the same manner.
- Schemas are widely accepted by most major parsing engines.
- Schemas allow you to data type with many different types. DTD only allows type content to be a string.

Now that you have had a brief look at the XSD schemas, the next section provides an in-depth look at schemas.

In-Depth Look at Schemas

One of the best ways to understand the XML schema language is to take a look at it; therefore, this section provides you with a brief example of a simple XML schema document followed by the XML document instance that conforms to the schema.

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace="http://www.wrox.com/books/xml"
  xmlns="http://www.wrox.com/books/xml">
<xsd:element name="books">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="book" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="title" type="xsd:string"/>
            <xsd:element name="author" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

Notice that schemas look similar to XML, and are usually longer than a DTD; typically, schemas are longer because they contain more information. Here is the XML document instance that conforms to the schema declaration.

```
<?xml version="1.0"?>
<books>
  <book>
    <title>XSLT Programmers Reference</title>
    <author>Michael Kay</author>
  </book>
</books>
```

Starting at the top with the preamble, the schema can be dissected as follows. A schema preamble is found within the element, schema. All XML schemas begin with the document element, schema. The first `xmlns` attribute is used to reference the namespace for the XML schema specification; it defines all the elements used to write a schema. The second `xmlns` attribute declares the namespace for the schema you are creating. Three letters is usually good for a namespace, but it can be longer. XML schemas can independently require elements and attributes to be qualified. The `elementFormDefault` attribute specifies whether or not elements need to be qualified with a namespace prefix. The default value is "unqualified." This schema, like most schemas, assigns the value of "qualified," which means that all locally declared elements must be qualified. This attribute also allows schemas to be used as the default schema for an XML document without having to qualify its elements. The `targetNamespace` attribute indicates the namespace and URI of the schema being defined.

> *The attribute targetNamespace is important because it's used to indicate this schema belongs to the same vocabulary as other schemas that reference the same namespace. This is how large vocabularies can be built, stringing them together with the schema keyword include.*

Now that you have a general understanding of the xsd:schema element, consider the following two constructs:

```
<xsd:element name="books">
<xsd:sequence>
```

The `xsd:element` uses the name attribute to define an element name (books); then, the sequence element is a compositor that tells the processor that the child elements nested with the sequence element must occur in that order when used as a part of an XML document instance.

XML Schema Datatypes

Datatypes provides document authors with a robust, extensible datatype system for XML. This datatype system is built on the idea of derivation. Beginning with one basic datatype, others are derived. In total, the datatypes specification defines 44 built-in datatypes (datatypes that are built into the specification) that you can use. In addition to these built-in datatypes, you can derive your own datatypes using techniques such as restricting the datatype, extending the datatype, adding datatypes, or allowing a datatype to consist of a list of datatypes.

XML Schema Usage Scenarios

There are many reasons document authors are turning to XML schema as their modeling language of choice. If you have a schema model, you can ensure that a document author follows it. This is important if you're defining an e-commerce application and you need to make sure that you receive exactly what you expect — nothing more and nothing less — when exchanging data. The schema model also ensures that data types are followed, such as rounding all prices to the second decimal place, for example. Another common usage for XML schema is to ensure that your XML data follows the document model before the data is sent to a transformation tool. For example, you may need to exchange data with your parent company, and because your parent company uses a legacy document model, your company uses different labeling (`bookPrice` versus `price`). In this case, you would need to transform your data so it conforms to the parent company's document model; however, before sending your XML data to be transformed, you want to be sure that it's valid because one error could throw off the transformation process. Another possible scenario is that you're asked to maintain a large collection of XML documents and then apply a style sheet to them to define the overall presentation (for example, for a CD-ROM or Web site). In this case, you need to make sure that each document follows the same document model. If

one document uses a para instead of a p element (the latter of which the style sheet expects), the desired style may not be applied. These are only a few scenarios that require the use of XML schema (or a schema alternative).

There are countless other scenarios that would warrant their use. The XML Schema Working Group carefully outlined several usage scenarios that it wanted to account for while designing XML schema. They are as follows:

- Publishing and syndication
- E-commerce transaction processing
- Supervisory control and data acquisition
- Traditional document authoring/editing governed by schema constraints
- Using schema to help query formulation and optimization
- Open and uniform transfer of data between applications, including databases
- Metadata interchange

As defined by the XML Schema Requirements Document, the previous usage scenarios were used to help shape and develop XML schema.

XSLT

Extensible Stylesheet Language Transformations (XSLT) is a language used for converting XML documents from one format to another. Although it can be applied in a variety of ways, XSLT enjoys two primary uses:

- Converting XML documents into HTML documents
- Converting XML documents into other XML documents

The first application — turning XML into HTML — is useful for building Web pages and other browser-based documents in XML. XML defines the content and structure of data, but it doesn't define the data's appearance. Using XSLT to generate HTML from XML is a fine way to separate content from appearance and to build generic documents that can be displayed however you want them displayed. You can also use Cascading Style Sheets (CSS) to layer appearance over XML content, but XSLT is more versatile than CSS and provides substantially more control over the output.

Here is how the transformation works: You feed a source XML document and an XSL style sheet that describes how the document is to be transformed to an XSLT processor. The XSLT processor, in turn, generates the output document using the rules in the style sheet. You see an in-depth discussion on XSLT and its usage in .NET in Chapter 7.

As mentioned earlier in this chapter, XSLT can also be used to convert XML document formats. Suppose company A expects XML invoices submitted by company B to conform to a particular format (that is, fit a particular schema), but company B already has an XML invoice format and doesn't want to change it to satisfy the whims of company A. Rather than lose company B's business, company A can use XSLT to convert invoices submitted by company B to company A's format. That way, both companies are happy, and neither has to go to extraordinary lengths to work with the other. XML-to-XML XSLT conversions are the cornerstone of middleware applications such as Microsoft BizTalk Server that automate business processes by orchestrating the flow of information.

XML DOM

The W3C has standardized an API for accessing XML documents known as XML DOM. The DOM API represents an XML document as a tree of nodes. Because an XML document is hierarchical in structure, you can build a tree of nodes and subnodes to represent an entire XML document. You can get to any arbitrary node by starting at the root node and traversing the child nodes of the root node. If you don't find the node you are looking for, you can traverse the grandchild nodes of the root node. You can continue this process until you find the node you are looking for.

The DOM API provides other services in additional to document traversal. You can find the full W3C XML DOM specification at `http://www.w3.org/DOM`. The following list shows some of the capabilities provided by the DOM API:

- Find the root node in an XML document.
- Find a list of elements with a given tag name.
- Get a list of children of a given node.
- Get the parent of a given node.
- Get the tag name of an element.
- Get the data associated with an element.
- Get a list of attributes of an element.
- Get the tag name of an attribute.
- Get the value of an attribute.
- Add, modify, or delete an element in the document.
- Add, modify, or delete an attribute in the document.
- Copy a node in a document (including subnodes).

The DOM API provides a rich set of functionality to programmers as is shown in the previous list. The .NET Framework provides excellent support for the XML DOM API through the classes contained in the namespace `System.Xml`, which you will see later in this book. The DOM API is well suited for traversing and modifying an XML document, but, it provides little support for finding an arbitrary element or attribute in a document. Fortunately another XML technology is available to provide this support: XML Path Language (XPath).

XPath

XML is technically limited in that it is impossible to query or navigate through an XML document using XML alone. XPath language overcomes this limitation. XPath is a navigational query language specified by the W3C for locating data within an XML document. You can use XPath to query an XML document much as you use SQL to query a database. An XPath query expression can select on document parts, or types, such as the document's elements, attributes, and text. It was created for use with XSLT and XPointer, as well as other components of XML such as the upcoming XQuery specification. All of these technologies require some mechanism that enables querying and navigation within the structure of an XML document.

The word *path* refers to XPath's use of a location path to locate the desired parts of an XML document. This concept is similar to the path used to locate a file in the directories of a file system, or the path specified in a URL in a Web browser to locate a specific page in a complex Web site. One of the most important uses of XPath is in conjunction with XSLT. For example, you can utilize XPath to query XML documents and then leverage XSLT to transform the resulting XML into an HTML document (for display in any format desired) or any other form of XML (for import into another program that may use a different set of XML tags). XPath is very powerful in that you can not only use it to query an XML document for a list of nodes matching a given criteria, but also apply Boolean operators, string functions, and arithmetic operators to XPath expressions to build extremely complex queries against an XML document. XPath also provides functions to do numeric evaluations, such as summations and rounding. You can find the full W3C XPath specification at `http://www.w3.org/TR/xpath`. The following list shows some of the capabilities of the XPath language:

- Find all children of the current node.
- Find all ancestor elements of the current context node with a specific tag.
- Find the last child element of the current node with a specific tag.
- Find the nth child element of the current context node with a given attribute.
- Find the first child element with a tag of `<tag1>` or `<tag2>`.
- Get all child nodes that do not have an element with a given attribute.
- Get the sum of all child nodes with a numeric element.
- Get the count of all child nodes.

The preceding list just scratches the surface of the capabilities available using XPath. Once again, the .NET Framework provides excellent built-in support for XPath queries against XML DOM documents and read-only XPath documents. You will see examples of this in later chapters.

SAX

In sharp contrast to XML DOM, the Simple API for XML (SAX) approaches its manipulation of a document as a stream of data parts instead of their aggregation. SAX requires the programmer to decide what nodes the application will recognize to trigger an event. DOM uses a parallel approach to the document, meaning it can access several different level nodes with one method. SAX navigates an XML document in serial, starting at the beginning and responding to its contents once for each node and in the order they appear in the document.

Because it has a considerably smaller memory footprint, SAX can make managing large documents (usually one measured in megabytes) and retrieving small amounts from them much easier and quicker. Because a SAX application approaches a document in search of nested messages for which it generates responses, aborting a load under SAX is easier than doing so under DOM. The speed by which you can find a certain type of node data in a large document is also improved.

XLink and XPointer

It's hard to imagine the World Wide Web without hyperlinks, and, of course, HTML documents excel at letting you link from one to another. How about XML? In XML, it turns out, you use XLinks and XPointers.

XLinks enables any element to become a link, not just a single element as with the HTML `<A>` element. That's a good thing because XML doesn't have a built-in `<A>` element. In XML, you define your own elements, and it only makes sense that you can define which of those represent links to other documents. In fact, XLinks are more powerful than simple hyperlinks. XLinks can be bidirectional, allowing the user to return after following a link. They can even be multidirectional — in fact, they can be sophisticated enough to point to the nearest mirror site from which a resource can be fetched.

XPointers, on the other hand, point not to a whole document, but to a part of a document. In fact, XPointers are smart enough to point to a specific element in a document, or the second instance of such an element, or any instance. They can even point to the first child element of another element, and so on. The idea is that XPointers are powerful enough to locate specific parts of another document without forcing you to add more markup to the target document.

On the other hand, the whole idea of XLinks and XPointers is relatively new and not fully implemented in any browser. Here are some XLink and XPointer references online that will provide you more information on these topics.

- `http://www.w3.org/TR/xlink/` — The W3C XLink page
- `http://www.w3.org/TR/xptr` — The W3C XPointer page

XQuery

XQuery (XML Query) is a language for finding and extracting (querying) data from XML documents. XQuery is a query language specification under development by the W3C that's designed to query collections of XML data — not just XML files, but anything that can appear as XML, including relational databases. Using XQuery, you can easily and efficiently extract information from native XML databases and relational databases. XQuery uses the structure of XML intelligently to express queries across all these kinds of data, whether physically stored in XML or viewed as XML via middleware.

XQuery makes heavy use of XPath. In fact, XQuery 1.0 and XPath 2.0 are under development by the same W3C working group, and their specifications are intertwined. XQuery 1.0 and XPath 2.0 share the same data model, the same functions, and the same syntax. Because the XQuery 1.0 specification is still in draft status, .NET Framework 2.0 does not provide support for XQuery 1.0 specification.

The XML Advantage

XML has had an impact across a broad range of areas. The following is a list of some of the factors that have influenced XML's adoption by a variety of organizations and individuals.

- XML files are human-readable. XML was designed as text so that, in the worst case, someone can always read it to figure out the content. Such is not the case with binary data formats.
- Widespread industry support exists for XML. Numerous tools and utilities are being provided with Web browsers, databases, and operating systems, making it easier and less expensive for small and medium-sized organizations to import and export data in XML format. For example, the .NET Framework has XML support available everywhere in the framework enabling the developers to easily and effectively utilize the power of the XML.
- Major relational databases such as SQL Server 20005 have the native capability to store, read and generate XML data.
- A large family of XML support technologies is available for the interpretation and transformation of XML data for Web page display and report generation.

Summary

Much more can certainly be written about XML and a number of books have done just that. This chapter just scratched the surface of XML by providing an overview of XML, highlighting important features related to XML. This chapter also discussed the related XML technologies used later in this book. To summarize this chapter:

- XML is extensible. It provides a specification for creating your own tags. XML is a metamarkup language.
- To be well formed, XML must essentially conform syntactically to the W3C specification, and all elements within the document must be children of one and only one document element.
- You have the ability to create your own tags, so make them meaningful. Because XML doesn't define any tags, creating tags that make sense to other developers is crucial.
- Namespaces provide a way to group elements and attributes into one vocabulary using a unique name. Using the xmlns attribute, a namespace prefix is bound to a unique namespace name.
- XML schemas offer developers a rich language to describe and define the structure, cardinality, datatypes, and overall content of their XML documents.
- Two object models exist for processing the content in any XML document: the DOM and SAX. The DOM allows random access and the capability to modify, delete, and replace nodes in the XML hierarchy. SAX provides a simple, efficient way to process large XML documents.
- XSLT provides a way to transform an XML document to another format such as HTML or another type of XML.
- XPath is a language that permits you to address the parts of an XML document.